SFMapReduce: An Optimized MapReduce Framework for Small Files

Fang Zhou Hai Pham Jianhui Yue Hao Zou Weikuan Yu
Auburn University
Auburn University, AL, 36849
{fzhou,htp0005,jianhui.yue,hzz0034,wkyu}@auburn.edu

***Abstract*—Hadoop, an open-source implementation of MapReduce, is widely used because of its ease of programming, scalability, and availability. With the explosive development of cloud computing, business and scientific applications increasingly take advantage of Hadoop. The sizes of files stored and processed in Hadoop are not bound to very large files anymore. However, Hadoop cannot provide stable and efficient services for small files at both storage and processing levels. To solve these problems, we propose an optimized MapReduce framework for small files, SFMapReduce. In SFMapReduce, we present two techniques, Small File Layout (SFLayout) and customized MapReduce (CMR). SFLayout is used to solve the memory problem and improve I/O performance in HDFS. CMR provides an interface for MapReduce so that SFMapReduce can process MapReduce with SFLayout efficiently. Our experimental results show that SFMapReduce decreases the memory pressure on the Hadoop NameNode, and provides better loading and retrieving throughput. On average, SFMapReduce achieves an improvement on MapReduce processing by 14.5 times and 20.8 times, compared with the original Hadoop and HAR layout.**

I. Introduction

Hadoop, an open-source implementation of MapReduce, has become an increasingly popular technology to process and analyze big data. It provides a reliable, scalable, distributed computing framework so users can focus more on the detailed applications.

The targeted data in Hadoop has been primarily very large files, such as large log files. Hadoop can provide a good performance of storing and processing for those large files. However, with the wide use of Hadoop, the large file is not the only kind of files stored in the Hadoop FileSystem (HDFS). In education, BlueSky is an E-learning cloud computing framework [1]. Most files stored in BlueSky are ppt files, whose file size normally starts from tens of Kilobytes to a few Megabytes. In climatology, The Earth System Grid (ESG) at LLNL stores over 27 Terabytes climate change files. The average size of these files is 61 Megabytes [2]. In the area of biology, some research applications create millions of files with an average size of less than 200 Kilobytes [3]. Similarly in astronomy, the files stored in Data Archive Server (DAS) at Fermi National Accelerator Laboratory (FNAL) are usually less than 1 Megabytes [4].

Small files appear not only at the storage level but also at the processing level. An increasing number of data analysis applications relies on Hadoop, such as remote sensing image analysis [5], web-scaled analysis for multimedia data mining [6], signal recognization [7], astronomy [8], meteorology [9], and climate data analysis [10]. Facebook also introduces the situation of data warehouse in [11], where they always need to process a lot of small files everyday.

The HDFS NameNode holds the metadata and block distribution in memory for each file. For large files, the memory pressure in NameNode is typically not a concern. However, when there are a lot of small files, the memory of NameNode would be exausted for storing the metadata and block information. Another problem is the costly block reports, by which DataNodes communicate with the NameNode. Heartbeat includes the node status and a block report provides the list of all blocks stored in a DataNode. If a large number of small files are stored in HDFS, the number of blocks stored in a DataNode would be very large. This situation increases the size of block reports sent from the DataNode, which incurs more overhead in the cluster. In addition, the performance of MapReduce programs degrades dramatically when their inputs include a large number of files. Plenty of creating and closing processes bring frequent context switchings, which lead to too much overhead.

In this paper, we propose a Small File MapReduce Framework (SFMapReduce) that can solve these problems systematically. Two techniques are introduced in our framework, which includes Small File Layout (SFLayout) and Customized MapReduce (CMR). SFLayout is an innovative file layout designed to use in HDFS for solving the storage problem of small files. SFLayout combines small files into an integrated file, which decreases the memory pressure of NameNode. In addition, we design several useful operators to manage the files stored in SFLayout. In order to process the files stored in the form of SFLayout, CMR is proposed to run the related MapReduce jobs. Moreover, CMR avoids the extra overhead and improves the MapReduce performance, compared with the conventional Hadoop. The cost of creating and closing a container keeps constant for any size of files. However, traditional Hadoop MapReduce generates containers for each small file. The cost of containers cannot be ignored in this situation. CMR is proposed to avoid the extra overhead and improve the MapReduce performance. CMR provides two customized components to transfer a SFLayout file to traditional Key/Value pairs in the processes of map and reduce phases. This approach reduces the overhead in running MapReduce

programs with lots of small files. CMR also contains a selector that is used to select specific files from SFLayout based on special conditions. SFMapReduce is designed to combine small files into an integrated file with a new layout and then run MapReduce programs based on the new layout.

The experiments' results illustrate the robustness of our framework. SFMapReduce provides better loading throughput than the original MapReduce and HAR file layout by 2.78x and 2.99x, respectively. At the same time, SFMapReduce provides better retrieving throughput than these two frameworks by 1.64x and 1.13x. Furthermore, SFMapReduce also outperforms the MapReduce's processing performance by 14.5x and 20.8x for different benchmarks on average.

In general, we have made four contributions in this paper: First, we identify the limitations of current frameworks for small files, such as memory pressure on NameNode, low loading and retrieving throughput, slow MapReduce processing. Second, we design and implement a new layout to store small files. This layout helps Hadoop solve the memory problem in NameNode and improve the loading and retrieving throughput. We also provide new API and operators to manage these small files. Third, we propose Customized MapReduce (CMR) inside Hadoop for running MapReduce programs efficiently on SFLayout. This technique enhances the MapReduce performance with the selective fuction which provides users great efficiency and flexibility. In the end, we conduct experiments to test the performance of SFMapReduce compared with the original Hadoop and some other frameworks. The experimental results show the effectiveness and efficiency of our framework.

The rest of this paper is organized as follows: Section II introduces the background and motivation in our research. Section III describes the design and framework of SFMapReduce. Section IV demonstrates experimental results, evaluation and analysis. Section V presents related work. We will conclude our paper in section VI.

II. Background and Motivation

In this section, we will first introduce the original mechanism of Hadoop Framework. Then we illustrate its problems of storing and processing small files. Next, to show the motivation for this work, we perform theoretical analyses to diagnose the reasons behind. And finally, we use experiments to demonstrate these inefficiency and ineffectiveness of Hadoop.

A. Overview of Hadoop Framework

Apache Hadoop contains three main components: Hadoop Common, HDFS, and Hadoop MapReduce. Particularly, Hadoop Common provides necessary utilities that support the other Hadoop modules. HDFS provides scalable, fault-tolerant, and distributed storage services. Hadoop MapReduce supports distributed and parallel processing of large scale of data.

1) HDFS: HDFS is comprised of a NameNode and several DataNodes. The NameNode takes the responsibility of managing all metadata information and responding to file operations, while each DataNode is responsible for storing replicated data blocks. Each data file is split into several blocks with the default size of 128MB, which are replicated in HDFS based on configuration. The NameNode regularly receives the heartbeat from DataNodes in a fix time interval - 3 seconds by default. On every heartbeat, each DataNode sends its status to the NameNode including the capacity, used space, remaining space and some other information. Upon receiving a heartbeat message, NameNode updates each DataNode's status and keeps it in memory. If it does not get any heartbeat from a DataNode in 10 minutes, it will remove that DataNode from the live node list and then add the node into the dead node list. Additionally, NameNode also stores each file and directory's metadata in memory. Thus in essence, the memory consumed in the NameNode is determined by the number of files stored in HDFS.

2) Hadoop MapReduce: The first design of Hadoop struggled with scalability because of its highly coupled structure. However, the second generation, named Yarn, significantly improves this property by seperating the JobTracker into two new components, the ResouceManager (RM) and the ApplicationMaster (AM). The old TaskTracker has been also supplanted by the NodeManager (NM). The AM and the NM use heartbeats to communicate with the RM. When a new job is submitted into Hadoop, the RM starts an AM on one slave node. Then this AM sends a resource request to the scheduler in the RM. After the scheduler allocates the resource, the AM needs to negotiate with NMs to start the containers. Once a container finishes, the AM and the NM can receive the notifications and use heartbeats to update the information to the RM.

B. Problems in Dealing with Small Files in Hadoop Framework

For storing and processing large files, Hadoop performs well. However, when dealing with the small files, the inefficiencies happen at both storage and processing levels.

1) The Problem of Storing Small Files: The NameNode holds each file's metadata in memory for quick response, including the file path, the block distribution, and so on. A file stored in HDFS correspondingly creates at least one block based on its size and the HDFS data block size. If the file size is less than the block size, only one block is created. When a large number of small files are stored in HDFS, lots of metadata are kept in memory on the NameNode. According to Hadoop's properties [12], in NameNode memory, a single directory object occupies 144 Bytes plus the length of its name; a single file object needs 112 Bytes plus the length of its name; and the size of each block object is 112 Bytes plus 24 Bytes times number of replicas. As a result, it is straightforward to speculate that if the number of files increases fast, the memory on NameNode would be exhausted. Another issue is the big size of block report which is used to coordinate block information of each DataNode with the one in NameNode. In detail, the default update interval for block report is 21,600 seconds. Although this interval is rather long, Hadoop cannot accept any read and write operation while updating the block information because of the data synchronization process. If there are huge number of small files in HDFS, the preparation of block report may take a few minutes.

2) The Problem of Processing Small Files: Before a container starts, the AM needs to negotiate with the RM and NMs. This doesn't cause much overhead when the scale of data input matches up with the container's computation capability, but it's not the case for small files in which the processing cost of computation is much higher. The reason is that containers cannot be re-used in Yarn, which means that the processes of creating and destroying containers happen continuously. The context switching, as a result, brings a lot of overhead. Moreover, one more issue is the I/O and communication overhead in the Reduce phase. In traditional MapReduce mode, Reduce tasks are used to gather different Map output files (MOFs) generated by Map tasks, compute the records based on keys, and then write the results to HDFS. However, by default, a small file only generates one InputSplit for the Map phase, which means only one Map task is launched for it. In this situation, the role of Reduce task is weakened since the operations in a Reduce task have been already done in the Map task. To sum up, the main overhead for processing a small file is imposed by storing MOF on the local disk for Map task and extracting MOF via network for Reduce task. For a large number of small files, this cost becomes much higher.

C. Demonstration of Hadoop Performance for Small Files

We have conducted some tests with small files to collect memory usage and MapReduce processing performance in Hadoop. The results show that Hadoop suffers a lot from storing and managing small files. The first experiment is to show the memory pressure in NameNode when there is a large number of small files stored in HDFS. The second one measures processing performance for a large number of small files compared with just only one large file, with the total data size being the same. We run these tests on a Hadoop cluster including 1 master node and 8 slave nodes. Each node in the cluster has two 2.67GHz hex-core Intel Xeon X5650 CPUs, 24GB memory and two 500 GB Western Digital SATA hard drives. Besides, Hadoop retains the default configuration. We use the Wordcount [13] benchmark and the test dataset is created randomly by Linux dictionary file[1]. We prepare 5 data sets for NameNode memory tests, which include 10,000, 100,000, 1,000,000, 10,000,000 and 100,000,000 files. The average file size in each set is 30 KB, which is typical in small files processing [1]. We also create other 10 testsets of small files from 1000 files to 45000 files for MapReduce tests. The average file size in these ones is 1023.65 KB. We also integrate these sets into ten large files as control groups.

1) Memory Pressure on NameNode for Small Files: We store each testset and the related integrated large file in an initialized Hadoop environment, respectively. Figure 1 shows the memory usage in the NameNode. When we store the amount of 3 TB small files into HDFS, the memory used by NameNode is more than 22 GB. According to the object size of file, directory, and

[1]Words file (/usr/share/dict/words) is the standard file on all Unix and Unix-like operating systems. It is used by spell-checking systems, such as ISpell, and normally includes 235,886 English words.

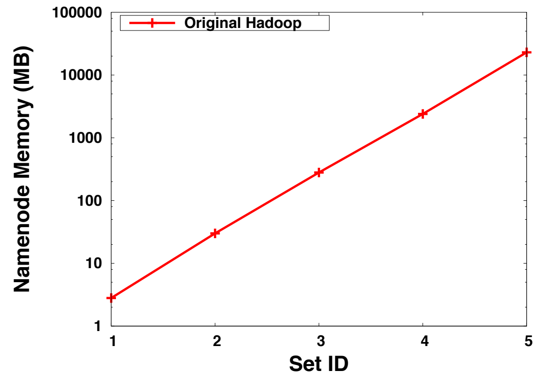


Fig. 1: NameNode Memory Usage

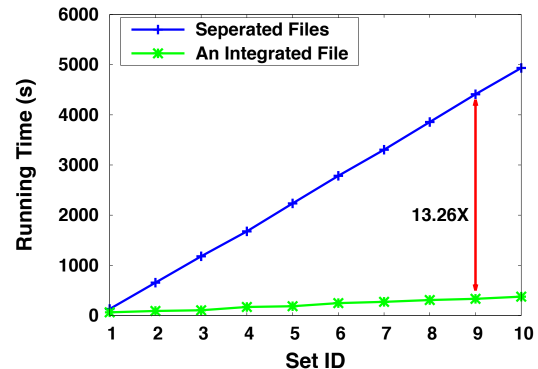


Fig. 2: Wordcount Performance with Two Different Inputs

block stored in memory on NameNode, we can deduce that more than 400 Million small files, whose total size is maybe about 12 TB, require nearly 90 GB of memory. Subsequently, although the total size of small files is not large, the memory can be used up quickly, causing the NameNode and then the cluster to shutdown.

2) Hadoop MapReduce's Processing Performance for Small Files: After storing the 10 testsets of small files and the corresponding integrated large files into HDFS, we test the MapReduce performance for these sets by running the original Wordcount provided by Hadoop. Thus there are two input modes in the tests: one is the directory storing all the small files; the other is the location of the respective large integrated file. As shown in Figure 2, when the number of small files increases, the gap between two input modes grows rapidly. In our small scale testing, the average performance of the set of small files is 10.49x lower than the that of the integrated file. Especially for testset 9, the performance of the integrated file is 13.26x higher than the other. We can reasonably project that the gap will continue to expand with the increasing number of files.

III. Design and Implementation

In order to solve the issues discussed in Section II, we design SFMapReduce, a new framework based on Hadoop/Yarn, In this section, we will describe the architecture of SFMapReduce,

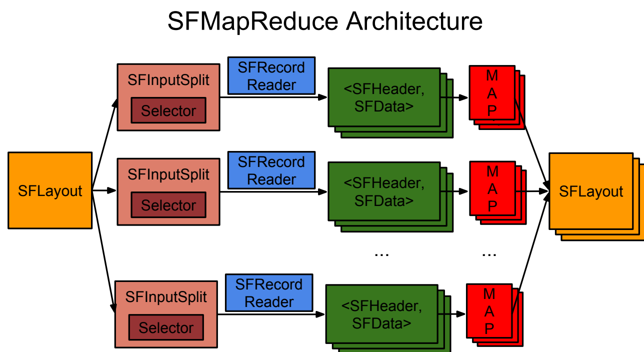


Fig. 3: SFMapReduce Architecture

the related techniques inside SFMapReduce, and the implementation details.

A. *Architecture of SFMapReduce*

Figure 3 shows the architecture of SFMapReduce. It is a pluggable framework based on current Hadoop/Yarn. Users can easily choose between SFMapReduce and original MapReduce framework. In SFMapReduce, the layout of the input file should be SFLayout. After a user submits a job to SFMapReduce, it first uses SFInputFormat class to produce CombineFileSplits so that Hadoop can start Map tasks. This step is similar to the original MapReduce in Hadoop. The biggest difference is that we use a new SFInputFormat class especially for SFLayout instead of the original InputFormat class in Hadoop. There is a Selector inside SFInputFormat, which helps generate InputSplits with selective Key/Value (KV) pairs based on specific conditions. After generating CombineFileSplits, each Map task starts processing with the corresponding KV pairs extracted from SFRecordReader. We design a new RecordReader because SFLayout is the new input format in our framework so that we cannot use original RecordReader to extract and parse SFLayout files. This step is also similar to the original RecordReader working in Hadoop. In SFMapReduce, MapReduce programs become Map-Only programs for decreasing overhead. That is why there is no Reduce task in the architecture. Finally, we get the result files in the form of SFLayout. This is the whole processing architecture for one job in our framework.

B. *SFLayout*

In this section, we first present the main structure of SFLayout, then introduce the design of related SFLayout operators.

1) The Main Structure of SFLayout: For better availability and efficacy, combining small files together into a large file is a very good and natural method. This is exactly what we have designed in the SFLayout. Figure 4 shows a general architecture for SFLayout. SFLayout is comprised of three components: SFIndex, SFHeader, and SFData. In detail, SFIndex contains the metadata of a small files and some other user-defined attributes. SFHeader and SFData are normally used for MapReduce processing. While SFHeader is the Key part, SFData is the Value part in the corresponding KV pair. SFLayout actually creates two separated files in HDFS. One file is called the

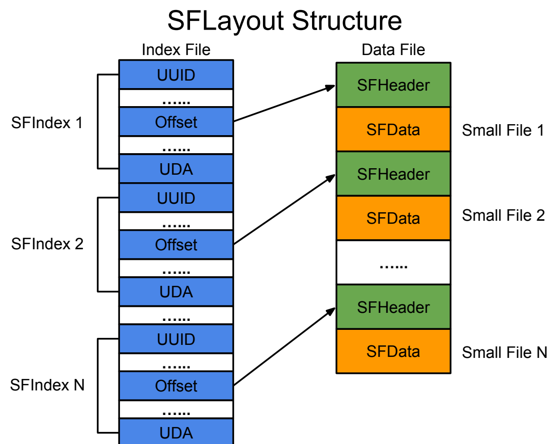


Fig. 4: SFLayout Structure

index file. The other is called the data file. For quick searching and easy maintenance, the index file is designed to only keep SFIndex so that the index file is not very large. The index file stores one file's SFIndex next to another one. In the data file, SFHeader and SFData are stored one by one. In the rest of this section, we will mainly introduce the detail of SFIndex, SFHeader, and SFData.

SFIndex: The information stored in SFIndex includes default file metadata and user-defined attributes. Default file metadata contains universally unique identifier (UUID), file name, file size, creation time, file owner, offset, and valid tag. The UUID is a 128-bit identifier, which is used to uniquely identify a file in the set of files. In our design, we choose the MAC address and the creation time to compute the UUID for each file. The file name, file size, creation time, and file owner are easily extracted from the file itself. Offset shows the starting position of the corresponding SFHeader in the data file. It is used for locating the KV pair position in the data file. And last, the valid tag shows the validation of the file.

SFHeader: SFHeader is used as Key for MapReduce processing so that it only needs to keep just enough identified information. Restrictly, the SFHeader is a subset of the SFIndex. In order to correctly identify each file in MapReduce processing, SFHeader at least stores the UUID and file name information. In addition, according to the detailed algorithms in MapReduce, users might want to add some more attributes in the SFHeader. For example, if they want to process an unstructured file in MapReduce, the SFHeader should include some more attributes that can be used to parse it in Map tasks.

SFData: SFData is stored next to the corresponding SFHeader in the data file. SFData only stores the file content itself in byte serialization. Note that the offset value in SFIndex points to the starting position of SFHeader in the data file.

2) The Design of SFLayout Operators: In our framework, we provide four operators to manage and maintain the SFLayout file. They include **Add**, **Remove**, **Update**, and **Get**. We describe the details of these operators as follows.

Add: Add is the most important operator for SFLayout. When a user wants to add local files to an SFLayout file in HDFS, she/he just simply runs the Add operator in the command line. Here, the user should show the source files or the directory in the local disk and the destination file path of the SFLayout file in HDFS. The framework extracts the metadata and user-defined attributes from the source file, then stores them to index file. Meanwhile, SFHeader and SFData are stored in the data file in the similar way.

Remove: HDFS does not support the update or modify operations so we cannot change the value of a file stored in HDFS. The valid tag in SFIndex helps our framework provide the elegant Remove function. In the valid tag, ”0” states the unavailability of the file, and ”1” shows the availability of the file. The default value is ”1”. If a user wants to delete the file from the SFLayout file, she/he just simply runs the Remove operator with the target file name on an SFLayout file. According to the file name, our framework starts to create a new index file where we only change the value of valid tag from 1 to 0, then delete the old index file and change the name to the original one. This operation does not take too much overhead because the index file is always very small compared with the data file. After the operation, although the space is still occupied, the removed file is transparent to the users.

Update: After a period of use, especially for some Remove operations, there may be a large number of gaps, which are worthless and ineffective in the SFLayout file The Update operator provides a method to erase these gaps in the SFLayout file. The framework creates a new SFLayout file without any gap and replaces the old one. It should be noted that HDFS cannot support update operation so once a file is stored in HDFS, we cannot change any bytes of the files. This is the reason that we create a new file in HDFS and replace the old file with the new one.

Get: Sometimes, users need to get some files from HDFS back to local disk. In order to meet this demand, we design the Get operator. We provide two access modes to get the files back from HDFS. One is sequential access and the other is random access. The sequential access is to copy all the files stored in SFLayout to the local disk. The random access is used to copy appointed files stored back based on specific conditions.

The operators discussed above are highly optimized because of minimizing the number of processes and the I/O overhead based on our design. These operators can provide good loading and retreving throughput, which are very common operations for processing small files.

C. Customized MapReduce

The original Hadoop MapReduce processing is not suitable for our framework because it still handles the traditional file layout. It is necessary for us to build new MapReduce components for SFLayout. In the following paragraphs, we present Customized MapReduce (CMR) technique.

1) Map-Only Application: As discussed above, only one Map task is created for a small file. And thus obviously, it can directly compute the final results without any help from Reduce task. So that is why there is only Map function existing in our SFMapReduce application. More specifically, in the traditional MapReduce applications, the Map task costs I/O and computing overhead when storing the MOF into the local disk. The Reduce task degrades the performance because of the extra cost of creating process and network communication. In our Map-Only application[2], we start the write input stream for the destination file in the *setup*() function and close it in the *cleanup*() function inside the Map class, which helps us finish the work faster than continue with running Reduce tasks.

2) SFInputFormat: In Hadoop, InputSplit is the minimum input unit for the Map task. The information kept by InputSplit includes file path, the data offset in the file, the data length,the list of nodes where the file is stored, and the information of related storage blocks. The number of InputSplits directly decides the number of Map tasks. Typically, the large file used in MapReduce programs produces more than one InputSplit. However, for the target small files, Hadoop normally generates one InputSplit for one file. This means Hadoop has to create a large number of Map tasks, *i.e.* processes. To continuously create and close a multitude of processes and context switchings incurs a lot of overhead, which really hurts the performance.

Design: In order to decrease the number of InputSplits, we choose CombineFileSplit as the InputSplit of Map tasks. CombineFileSplit is a built-in InputSplit in Hadoop, which is a set of input files. In Hadoop, InputFormat is used to generate InputSplits. InputFormat describes the input-specification for a MapReduce job. However, InputFormats provided by Hadoop cannot be used on the SFLayout file. For parsing SFLayout correctly and decreasing the number of Map tasks, we design SFInputFormat as a component in CMR. SFInputFormat is used to parse and collect the information stored in the SFLayout file and generate a list of CombineFileSplits.

Selector: For the flexibility and functionality, we design a *selector* component in the SFInputFormat. To be specific, we add a *selector()* function in the SFInputFormat class. According to the SFIndex in the input file, the *selector()* function can easily get the information of each small file. Before adding a small file into the InputSplit, we need to first consider the returned value of the *selector()*. If the file's information accords with the conditions, then *selector()* returns true and this file is added into the InputSplit. If not, false is returned and this file is skipped. With the help of the *selector()* function, users can run the jobs flexibly and selectively. In our framework, the default *selector()* function always returns true. Users can write any complex judgment inside the code based on their demands.

Partition Algorithm: The partition algorithm in SFInputFormat is the most substantial because it is used to generate the CombineFileSplits. A very important threshold, SFSplitSize, is used to limit the total size of the files contained by a CombineFileSplit, which decides how much data one InputSplit can contain. Our partition algorithm is shown in detail in

[2]Map-Only Application is our recommendation for processing small files. Actually, SFMapReduce framework can process all kinds of MapReduce programs.

Algorithm 1 SFInputFormat Partition Algorithm

1: Input: SFLayout: $sfLayout$, the threshold of split size: $size_limit$
2: Output: A List of InputSplit: lis
3: Initialization
4: **while** sfLayout.hasNext() **do**
5: **if** InputSplit is $=$ NULL **then**
6: $is \leftarrow NewCombineFileSplit()$
7: $is.size \leftarrow 0$
8: **end if**
9: SFIndex $sfIndex \leftarrow sfLayout.getIndex()$
10: **if** Selector($sfindex$) **then**
11: Add the file path, offset, length, and block information into the CombineFileSplit is
12: is.size += sfIndex.getLength()
13: **if** is.size $>$ size_limit **then**
14: lis.add(is)
15: $is \leftarrow null$
16: **end if**
17: **end if**
18: **end while**
19: **if** is $!=$ NULL **then**
20: lis.add(is)
21: **end if**

Algorithm 1. First, the algorithm checks whether we have initialized the CombineFileSplit. If it is existed, then continue with running; if not, initialize a new CombineFileSplit (lines 5-8). The next step is to get an SFIndex from the SFLayout (line 9). After getting the SFIndex, we first use the *selector()* function to check whether this file meets with the users' requirements. (line 10). If the result is true, then we extract the necessary information and store it in the CombineFileSplit (line 11). Then we update the size of the current CombineFileSplit (line 12) and continue to check if the size reaches the limit of the SFSplitSize (lines 13-16). If the size is larger than the SFSplitSize, then we add the CombineFileSplit into the list and clear the InputSplit variable. Finally, the function returns the list of CombineFileSplits. From the algorithm, we can find out that SFInputFormat creates a much less number of InputSplits than traditional method in Haddop. This can significantly decrease the number of Map tasks, which greatly improves the performance of our framework.

3) SFRecordReader: RecordReader is a very important component in Hadoop MapReduce. It takes the responsibility to transfer the byte stream, provided by InputSplit, to the record stream (KV pairs), used by Map tasks. However, the original RecordReader cannot support the SFLayout. In order to solve this problem, we design a new RecordReader named SFRecordReader. In the new SFRecordReader, we override the related functions to provide the necessary services to Map tasks. Responding to the requests from Map tasks, SFRecordReader returns SFHeader as Key and SFData as Value.

TABLE I: List of key Hadoop configuration parameters.

Parameter Name	Value
yarn.nodemanager.resource.memory-mb	22528 MB
yarn.scheduler.maximum-allocation-mb	6144 MB
yarn.scheduler.minimum-allocation-mb	2048 MB
yarn.nodemanager.vmem-pmem-ratio	2.1
MapReduce.map.java.opts	2048 MB
MapReduce.reduce.java.opts	2048 MB
MapReduce.task.io.sort.factor	100
dfs.block.size	128 MB
dfs.replication	3
io.file.buffer.size	8 MB

D. Implementation

We have implemented our framework based on Hadoop/Yarn 2.6. For the layout, we implement the SFLayout class including four main operator functions, the SFIndex class and SFData class using Java. For CMR, we create the SFInputFormat extended from CombineFileInputFormat and SFRecordReader class derived from existing classes in Hadoop and override the related functions.

Our framework is implemented as a plugin so users can directly use our framework without any changes in the currently using Hadoop clusters. This is also a big advantage of our framework.

IV. Evaluation

In this section, we evaluate the effectiveness of SFMapReduce compared with some existing frameworks. We will first describe the experiemental environment, then provide the results with analyses

A. Experimental Environment

1) Cluster Setup: Our private cloud is comprised of 17 computer servers, each of which has a 2.67 GHz hex-core Intel Xeon X5650 CPU, 24 GB memory and two 500 GB Western Digital SATA hard drives. The machines are connected through 1 Gigabit Ethernet. In experiments, we create a Hadoop cluster of these 17 nodes.

2) Hadoop Setup: We use Hadoop/Yarn-v2.6.0 as the code base with JDK 1.7. One node is dedicated as the ResourceManager and the Namenode of Yarn and HDFS. As a result, in our cluster, we have 16 slave nodes. The key configurations of Hadoop/Yarn we use can be seen in the Table I.

3) Benchmark: In the experiments, we choose the most representative program: Wordcount, TeraSort, and Grep to test the performance. It should be noted that these three benchmarks are Map-Only applications. The reason why we use the Map-Only program has been discussed in the Section III-C. The detail of these benchmarks can be seen in [13], [14], and [15].

4) Alternative Solution Setup: Instead of comparing with the original Hadoop, we also compare our SFMapReduce with one solution inside Hadoop (HAR).

Hadoop Archive (HAR): HAR is a special file layout built in Hadoop for small files. A HAR file usually ends with a .har extension. It includes a Masterindex, Index, and the data file including all the small files. In our test, we use the data

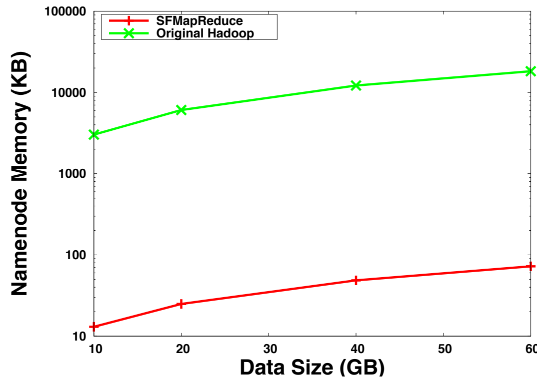


Fig. 5: Memory Usage in Namenode

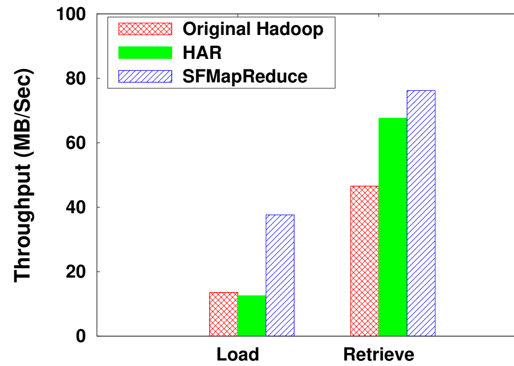


Fig. 6: Loading and Retriving Throughput

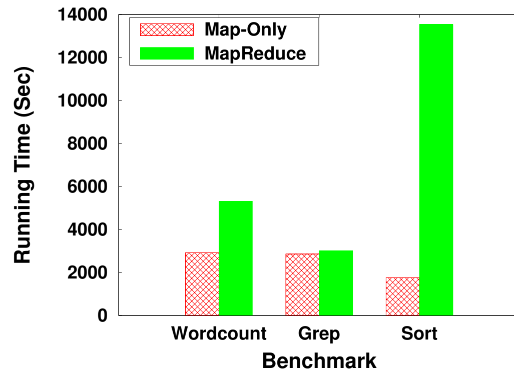


Fig. 7: Performance of Map-Only and MapReduce jobs

prepared to create the corresponding HAR files, then use these files as the input for related benchmarks.

5) Data Preparation: For Wordcount and Grep, we randomly extract part of Shakespeare Complete Works [16] to create lots of files with different sizes. For TeraSort, we use TeraGen to randomly create the number of lines in the result. After preparing the data, we divide the small files into 4 sets for each benchmark. The total sizes of the sets are 10 GB, 20 GB, 40 GB, and 60 GB, respectively.

B. Memory Utilization

We compare the memory utilization of SFMapReduce with original Hadoop, when we store the four data sets into them. As shown in figure 5, SFMapReduce, by employing SFLayout, reduces the memory usage on Namenode by 248x compared with the original Hadoop. In the original Hadoop, one small file consumes one file entry and one data block entry in the Namenode's memory. However, no matter how many small files we store in SFMapReduce, it only stores two files: the index file and the data file. So SFMapReduce only occupies two file entries and related data block entries. The number of data block entries is equal to the total size of files divided by HDFS data block size.

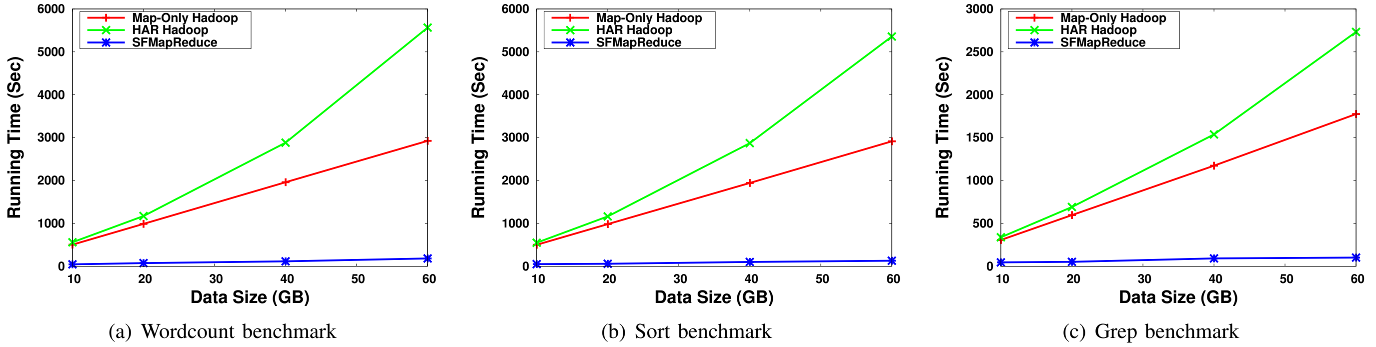
C. Data Loading and Retrieving

For processing small files, data loading and retrieving are very common and important operations. Loading time and retrieving time are very significant performance indices, because users need to load data for processing (from disk to HDFS) and to retrieve data back for observing results. We conduct experiments to test the throughput of loading and retrieving data in SFMapReduce, compared with the original Hadoop and HAR. For these tests, we use the Hadoop cluster prepared, and the scales of dataset are 10 GB, 20 GB, 40 GB, and 60 GB as introduced the in the *Data Preparation* section above. In order to make the results more reliable, we calculate the average throughput based on 5 experiments. The final results can be seen in Figure 6.

On average, the loading throughput of SFMapReduce is faster than the original Hadoop by 2.78x and HAR by 2.99x; the retrieving throughput is better than the original Hadoop by 1.64x and HAR by 1.13x. In term of loading, when a file is added into HDFS, the original Hadoop creates and then closes a data inputstream for the file. If a large number of files needs to be loaded into HDFS, repeating of creating and closing streams brings a lot of overhead. In addition, Namenode needs to allocate the nodes storing replicas for each file, which leads to additional overhead when there are a lot of small files.

In the case of HAR, creating an HAR file is actually a MapReduce job. This means, before we start creating it, we should first load the files into HDFS. In fact, the total loading time of HAR layout is equal to the loading time of original Hadoop plus the job time of creating HAR file.

In our new approach, SFMapReduce starts an inputstream to HDFS for the index file and data file at first. Then the inputstream continues to be used until all the files have been inserted into HDFS. In the process, we avoid the repetition of creating and closing the inputstream, hence yielding a very good performance. For retrieving data, Hadoop does it a similar way to loading data. HAR has a good retrieving throughput because Hadoop can sequentially retrieve data from a large HAR file instead of reading each file continuously. Last, our SFMapReduce still outperforms the others by eliminating the repetition of creating and closing the streams as in loading operation. To sum up, these experiments show the advantages of the loading and retrieving operations in the SFMapReduce.



(a) Wordcount benchmark (b) Sort benchmark (c) Grep benchmark

Fig. 8: Job Running Time

D. Map-Only Application Performance

We start our experiments to prove the Map-Only application can provide better performance than the original MapReduce application, as discussed in Section III-C. For comprehensiveness, we use the three benchmarks for testing and change slightly MapReduce programs to Map-Only ones for these benchmarks. The input size of these experiments are all 60 GB. For stability, we run each test 5 times and collect the average running time. As shown in Figure 7, the average running time of Map-Only jobs is less than original MapReduce jobs by 3.52x. Specifically, Map-Only jobs have 1.81x, 1.05x, 7.68x better performance than the original MapReduce jobs, for Wordcount, Grep, and Sort. From the results, we can find that the more network communication a MapReduce job makes, the worse performance is. The reason is the benefit of Map-Only jobs comes from avoiding the overhead from the intermediate data and the Reduce phase. If the intermediate data of MapReduce jobs is very small, like Grep job, the performance between these two kinds of jobs is very similar. Based on these experiments, we choose Map-Only jobs instead of traditional MapReduce jobs in SFMapReduce. Although we choose Map-Only jobs in SFMapReduce, it does not show the incapability of MapReduce jobs in our framework. If users need to run MapReduce jobs in some special condition, SFMapReduce can perform it very well. In the rest of this paper, we use Map-Only jobs to replace the original MapReduce jobs.

E. Tuning the Size of SFSplitSize

In this section, in order to get the best performance of the SFMapReduce, we want to tune the size of the SFSplitSize, which directly decides the size of an InputSplit. For split size, we should make a balance between data access and parallelism. We use Wordcount benchmark to test SFMapReduce performance with different sizes of SFSplitSize from 32 MB to 512 MB. The results are shown in Figure 9. It is obvious that when the value of SFSplitSize is equal to 128 MB, SFMapReduce performs the best. In our framework, HDFS block size is 128 MB. This means when the SFSplitSize is near or equal to HDFS block size, we can achieve the best performance. The reason is that if the value of SFSplitSize is too small, then data access overhead is increased; if it is too large, the task processing parallism is degraded. In the rest part of the evaluation, we choose 128 MB as the value of SFSplitSize.

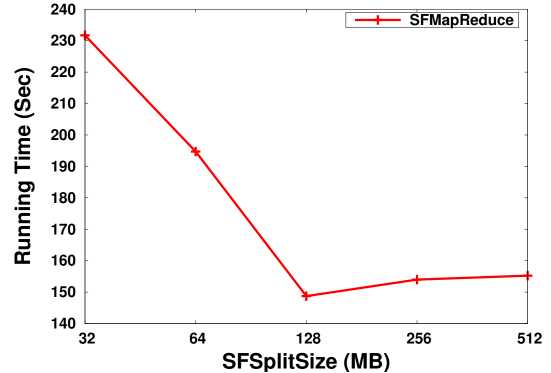


Fig. 9: The Impact of SFSplitSize

F. Overall Performance

In this section, we test the SFMapReduce's processing performance, compared with other frameworks. In SFMapReduce, we set the SFSplitSize to 128 MB based on the tuning results shown in the previous section. The scales of data set are 10 GB, 20 GB, 40 GB, and 60 GB, respectively. We conduct the experiments 5 times and collect the average running times. The test frameworks include original MapReduce Hadoop, HAR Hadoop and SFMapReduce. In these frameworks, we all use Map-Only Wordcount, Sort, and Grep benchmark. Map-Only Hadoop means the original Hadoop with Map-Only jobs. HAR Hadoop means the Hadoop processing with the HAR layout. As shown in Figure 8, on average, the performance of SFMapReduce overperforms the original Hadoop by 14.5x and HAR Hadoop by 20.8x. For Wordcount benchmark, the performance of SFMapReduce overperforms the Hadoop by 14.4x and HAR Hadoop by 21.0x; for Sort, SFMapReduce is better than the original Hadoop by 12.1x and HAR Hadoop by 16.1x; for Grep, our framework runs faster than the original Hadoop by 17.1x and HAR Hadoop by 25.2x; Moreover, Figure 10 shows the efficiency of our framework from the perspective of processing throughput on average. The original Hadoop and HAR Hadoop do not do any optimization in the MapReduce processing. On the contrary, SFMapReduce provides the best performance, because it avoids the unnecessary overhead and combine multiple

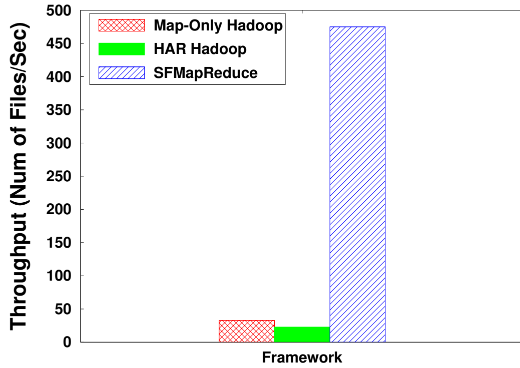


Fig. 10: File Throughput

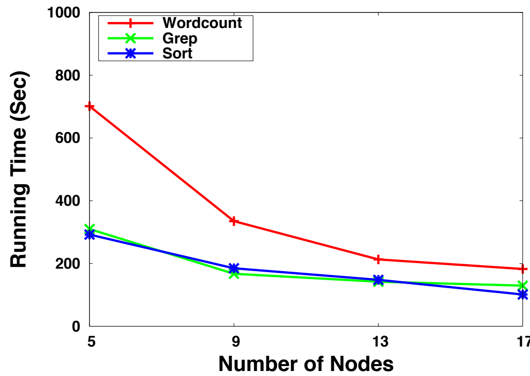


Fig. 11: SFMapReduce Scalability with Increasing Number of Nodes

small files together into one InputSplit. Unlike the conventional Hadoop, SFMapReduce decreases the number of Map tasks so that reduces the extra repetitive overheads incurred by creating and closing related containers. Likewise, the SFLayout and related CMR work together very well. For the KV pairs, CMR can efficiently get them from SFLayout; for the input splits, SFMapReduce makes a balance between parallism and access overhead. The whole processes are direct and highly efficient.

G. Scalability

After talking about the overall performance of SFMapReduce, we continue to show the scalability of SFMapReduce. We test scalability from two different perspectives. One is the strong scaling which is the performance of a framework running on an increasing number of computation resources, i.e. nodes. If a framework has a good strong scaling, the performance should become better with the increasing number of nodes while the problem size stays the same. The other is the weak scaling which is the efficiency of a framework when we add more computing elements to solve the additional input, so that each element of input is not so large to fit with each computing element. If a framework has a good weak scaling, the performance should stay similar when the number of nodes increases along with the input size. In experiments, we first test the strong scaling with the different benchmarks on the clusters of 5 nodes, 9 nodes, 13 nodes, and 17 nodes, respectively. Then we test the weak scaling of SFMapReduce on the similar

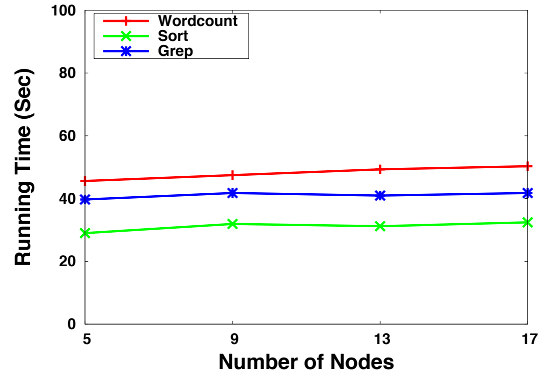


Fig. 12: SFMapReduce Scalability with Increasing Data Size

clusters and benchmarks, but the input varies with number of nodes so that each node deals with only 1 GB data. To get each result, we collect the average time based on 5 different runnings.

As shown in Figure 11, on average, SFMapReduce on the cluster of 17 nodes can keep an improvement of 2.93x over the cluster of 9 nodes and 1.76x over the cluster of 5 nodes. This shows the strong scaling of SFMapReduce. In Figure 12, we can see, the performances on different clusters keep unchanged. This also proves that SFMapReduce has very good weak scaling.

V. Related Work

HAR, SequenceFile, and MapFile are the pre-built techniques in HDFS. They have very similar ideas of combining the files into one file. However, they focus on solving the problem in HDFS while ignoring the optimization in MapReduce processing. Our experiments also showed that their performances were not as good as the original Hadoop.

Chen *et al.* [17] proposed two techniques to solve the problem of small files in HDFS. The first one was very similar to our SFLayout. They integrated the small files into a large file and built an index for them. The second one was building a cache level beetween HDFS and users. This cache could help improve the performance of HDFS's read and write operations. However, this paper only focused on how to improve the I/O performance of HDFS. They did not give a solution on how to use it effectively in the MapReduce processing.

Zhang *et al.* [18] introduced a very similar research with [17]. The authors built a non-blocking I/O to merge small files into a large file in HDFS. However, they did not consider how to process the data stored in HDFS, either.

Mackey *et al.* [3] used the HAR file layout to solve the small files problem. They focused on how to schedule jobs in Hadoop cluster with quota policy. The HAR file layout is not a flexible and powerful layout. They did not provide good solutions for consecutive MapReduce processing.

Dong *et al.* [19] proposed a case study researching on small files problem in HDFS. This paper has provided a review on many techniques and related discussions. In this paper, the authors provided a technique to integrate small files into a

large file in HDFS. It was very similar to the other papers. In addition, the authors provided a prefetching technique to support the functions needed in their real education system. [20] was the authors' subsequent research. In this paper, the authors divided the files into three categories: structurally-related files, logically-related files, and independent files. For the first two types, they designed the special storing solutions. They also introduced three-level prefetching and caching strategies. However, both of the papers lost the consideration about how to run MapReduce jobs in the new Layout. They only focused on how to meet with the demands of a concrete system.

Yin *et al.* [21] proposed OPASS - a useful method for dealing with data read operations in HDFS which is relevant to our framework's technique of I/O optimization. However, there are two big differences between OPASS and SFMapReduce. First, OPASS did not take any small files into account like ours. Second, OPASS, unlike SFMapReduce which reorganizes the small files in to big chunks for facilitating the tasks dealing with huge number of them, just used HDFS directly.

Dittrich *et al.* [22] presented HAIL (Hadoop Aggressive Indexing Library). HAIL optimized the loading operation and MapReduce's processing performance in Hadoop. However, they did not focus on small files so that the related work cannot solve any problems we mentioned in this paper.

Jindal *et al.* [23] proposed the Trojan Data Layout. The authors designed this new data layout for each data block and related replicas. It was powerful and efficient. However, the objective of this layout was to increase the I/O speed in Hadoop. This technique did not solve the problems for small files as well.

VI. Conclusion and Future Work

Hadoop is a powerful and widely used framework to handle large scale of data. Users and developers can easily use Hadoop to parallelize the processes of data in an available and scalable cloud environment. The demands and requirements vary dramatically in the practical world. One of the most significant demands is to add features to efficiently store and process small files in Hadoop. As we discussed in background section, both HDFS and MapReduce in the original Hadoop cannot support small files well. In order to solve these problems, we propose the SFMapReduce framework built on top of Hadoop. In our framework, we propose two techniques, SFLayout and CMR, which help provide better storage and processing services. We also show that SFMapReduce solves the memory pressure on the Namenode and has better performance than the original Hadoop by 14.5x and HAR layout by 20.8x on average.

VII. Acknowledgment

This work is funded in part by an Alabama Innovation Award, and by National Science Foundation awards 1059376 and 1432892. The authors are very thankful to anonymous reviewers for their invaluable feedback.

References

[1] B. Dong, Q. Zheng, M. Qiao, J. Shu, and J. Yang, "Bluesky cloud framework: an e-learning framework embracing cloud computing," in Cloud Computing. Springer, 2009, pp. 577–582.

[2] A. Chervenak, J. M. Schopf, L. Pearlman, M.-H. Su, S. Bharathi, L. Cinquini, M. D'Arcy, N. Miller, and D. Bernholdt, "Monitoring the earth system grid with mds4," in e-Science and Grid Computing, 2006. e-Science'06. Second IEEE International Conference on. IEEE, 2006, pp. 69–69.

[3] G. Mackey, S. Sehrish, and J. Wang, "Improving metadata management for small files in hdfs," in Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on. IEEE, 2009, pp. 1–4.

[4] E. H. Neilsen Jr, "The sloan digital sky survey data archive server," Computing in Science and Engineering, vol. 10, no. 1, pp. 13–17, 2008.

[5] M. H. Almeer, "Hadoop mapreduce for remote sensing image analysis," International Journal of Emerging Technology and Advanced Engineering, vol. 2, no. 4, pp. 443–451, 2012.

[6] B. White, T. Yeh, J. Lin, and L. Davis, "Web-scale computer vision using mapreduce for multimedia data mining," in Proceedings of the Tenth International Workshop on Multimedia Data Mining. ACM, 2010, p. 9.

[7] F. Wang and M. Liao, "A map-reduce based fast speaker recognition," in Information, Communications and Signal Processing (ICICS) 2013 9th International Conference on. IEEE, 2013, pp. 1–5.

[8] K. Wiley, A. Connolly, J. Gardner, S. Krughoff, M. Balazinska, B. Howe, Y. Kwon, and Y. Bu, "Astronomy in the cloud: using mapreduce for image co-addition," Astronomy, vol. 123, no. 901, pp. 366–380, 2011.

[9] W. Fang, V. Sheng, X. Wen, and W. Pan, "Meteorological data analysis using mapreduce," The Scientific World Journal, vol. 2014, 2014.

[10] D. Q. Duffy, J. L. Schnase, J. H. Thompson, S. M. Freeman, and T. L. Clune, "Preliminary evaluation of mapreduce for high-performance climate data analysis," 2012.

[11] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu, "Data warehousing and analytics infrastructure at facebook," in Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010, pp. 1013–1020.

[12] "Hadoop-1687," http://issues.apache.org/jira/browse/HADOOP-1687.

[13] "Hadoop Wordcount WIKI," http://wiki.apache.org/hadoop/WordCount.

[14] "Hadoop Grep Wiki," http://wiki.apache.org/hadoop/Grep.

[15] T. White, Hadoop: The definitive guide. " O'Reilly Media, Inc.", 2012.

[16] "Shakespeare Complete Works," http://www.gutenberg.org/ebooks/100.

[17] J. Chen, D. Wang, L. Fu, and W. Zhao, "An improved small file processing method for hdfs," International Journal of Digital Content Technology and its Applications, vol. 6, no. 20, pp. 296–304, 2012.

[18] Y. Zhang and D. Liu, "Improving the efficiency of storing for small files in hdfs," in Computer Science & Service System (CSSS), 2012 International Conference on. IEEE, 2012, pp. 2239–2242.

[19] B. Dong, J. Qiu, Q. Zheng, X. Zhong, J. Li, and Y. Li, "A novel approach to improving the efficiency of storing and accessing small files on hadoop: a case study by powerpoint files," in Services Computing (SCC), 2010 IEEE International Conference on. IEEE, 2010, pp. 65–72.

[20] B. Dong, Q. Zheng, F. Tian, K.-M. Chao, R. Ma, and R. Anane, "An optimized approach for storing and accessing small files on cloud storage," Journal of Network and Computer Applications, vol. 35, no. 6, pp. 1847–1862, 2012.

[21] J. W. Jiangling Yin, D. H. Jian Zhou, Tyler Lukasiewicz, and J. Zhang, "Opass: Analysis and optimization of parallel data access on distributed file systems," in IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2015 IEEE. IEEE, 2015.

[22] J. Dittrich, J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad, "Only aggressive elephants are fast elephants," Proceedings of the VLDB Endowment, vol. 5, no. 11, pp. 1591–1602, 2012.

[23] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich, "Trojan data layouts: right shoes for a running elephant," in Proceedings of the 2nd ACM Symposium on Cloud Computing. ACM, 2011, p. 21.